

Module 11 Homework - Graphs

- 1) Implement a non-directional edge-weighted graph.
- 2) Add graph traversal algorithms to solve a few types of optimal path problems.

As always, use TDD here - write unittests first, then implement the methods in your **Graph** class.

Part 0 - Unittests

It will be helpful to have an image in mind when writing your tests. Look up 5 cities, and sketch an ascii-art graph of those 5, using distances as edge weights (don't use any of the 5 below):

```

                2040 mi                980 miles
Seattle ----- Chicago ----- Boston
|
|
| 1140 miles
|
|
Los Angeles ----- New Orleans
                1890 miles
```

Which and how many edges you draw aren't terribly important, but make sure your map is connected - it should be possible to reach any vertex from any vertex.

Store this graph in a `setUp()` method in your unittest classes, so you only have to define it once ([docs](#)).

Functionality

At minimum, support:

- `add_vertex(v)`
- `remove_vertex(v)`
- `add_edge(u, v, wt)`
- `remove_edge(u, v, wt)`
- `nbrs(v)` - returns an iterator over neighbors of `v`:

```
>>> for nbr in g.nbrs('Boston'):
>>>     print(nbr)
Chicago
New Orleans
```

You don't need to worry about multiple edges between the same two vertices (e.g. two edges between A and B with different weights) - we're only considering simple graphs.

Graph Traversal Algorithms

Implement algorithms to solve each of the problems below. The unittest skeleton code includes `# TODO` comments for you to denote which algorithm you are using for each problem and why. Each algorithm takes a `city` as an input:

1. `fewest_flights(city)` - finds how to get from `city` to **any other city in the graph** with the **fewest number of flights** - this is for busy users who need to get from city to city quickly, and cannot waste time in extraneous layovers.
2. `shortest_path(city)` - finds how to get from `city` to **any other city in the graph** with the **fewest number of miles travelled** - this is for environmentally concious users who want to use the least amount of fuel to get between two cities.
3. `minimum_salt(city)` - connects `city` to **every other city in the graph** with the **fewest total number of miles** - this tree would allow us to keep all cities connected in the winter with the least amount of salt used on roads.

Each method should return two items - a dictionary-tree showing traversal order, and a dictionary of `vertex:distance` pairs that describes the relevant distance used to add that vertex to the previous tree (the distance from source in algorithm 2, and the length of the edge in algorithm 3)

When testing these algorithms, **don't test that they give a fixed tree** - each problem may not have a unique answer. Instead, test what the algorithms guarantee. For instance, for part 3, test that the total distance in your returned tree is x miles, rather than that `Boston` is the parent of `Chicago`.

Submission

At a minimum, submit the following files:

- `Graph.py`
- `TestGraph.py`

Students must submit individually by 11:59 PM EST **Friday, Dec 9 2022**.

Grading

This assignment is 100% manually graded.

Feedback

If you have any feedback on this assignment, please leave it [here](#).